GIER—A DANISH COMPUTER OF MEDIUM SIZE

BY

C. GRAM, O. HESTVIK, H. ISAKSSON, P. T. JACOBSEN, J. JENSEN, P. NAUR,
B. S. PETERSEN, AND B. SVEJGAARD

Reprinted from IEEE TRANSACTIONS
ON *ELECTRONIC COMPUTERS*
Volume EC-12, Number 5, December, 1963

PRINTED IN THE U.S.A.

GIER—A Danish Computer of Medium Size*

C. GRAM†, O. HESTVIK‡, H. ISAKSSON†, MEMBER, IEEE, P. T. JACOBSEN‡, J. JENSEN†, P. NAUR†, B. S. PETERSEN†, ASSOCIATE MEMBER, IEEE, AND B. SVEJGAARD§

Summary—**This paper gives a brief review of the design of the machine GIER (42-bit words, 1-k core store, 12-k drum store, 50-microsecond fixed point, 100-microsecond floating, add time) and its programming systems. The principal subjects are: The order structure, the operating system, the ALGOL 60 system, an evaluation of the order structure, the hardware organization, and the latest hardware extensions, including a hybrid computer system.**

I. INTRODUCTION

THE GIER is a computer originally developed as a joint project of the Geodetic Institute and the Danish Institute of Computing Machinery (Regnecentralen), both of Copenhagen, with the purpose of providing a tool for solving problems of geodesy. However, subsequent to the completion of the prototype in late 1961 a production was taken up at Regnecentralen and by October, 1963, a total of 15 GIERs were in use in research and educational centers in Denmark, Norway, Germany, and France.

The present report reviews the highlights of the machine with regard to logical structure, software, and hardware. Sections II to IV, by Gram and Svejgaard, describe the order structure and operating system, stressing particularly the extensive address modification facilities, the handling of conditional instructions, and the use of marks attached to data. Section V, by Naur, describes the design of the ALGOL 60 system written for the machine, giving particular attention to the solutions adopted for overcoming the inconveniences of the two levels of store. In Section VI Jensen evaluates the order structure from the point of view of the compiler writer, starting from the techniques actually employed in the ALGOL compiler. Section VII, by Isaksson, describes the realization of the structure in hardware, in particular the general register transport organization and the system of microprograms used in executing the machine instructions. Finally in Section VIII, by Petersen, Jacobsen, and Hestvik, the latest hardware extensions of the machine, including those for controlling real-time processing, are discussed.

II. STRUCTURE OF GIER (FROM THE PROGRAMMER'S POINT OF VIEW)

A. Storage

GIER is a binary one-address computer with fixed word length. The *ferrite core store* is rather small, with only 1024 cells, but is backed up by a *magnetic drum store* with 12,800 cells, and it is possible to extend the store with a 4096-word buffer store for external units and with a 25,600-word drum (see Section VIII). Also, commmunication with the drum store may be very fast if the possibility for simultaneous drum transfer and calculation is exploited: A drum track of 40 words is transferred in 20 msec of which about 19 msec can be utilized for simultaneous calculations.

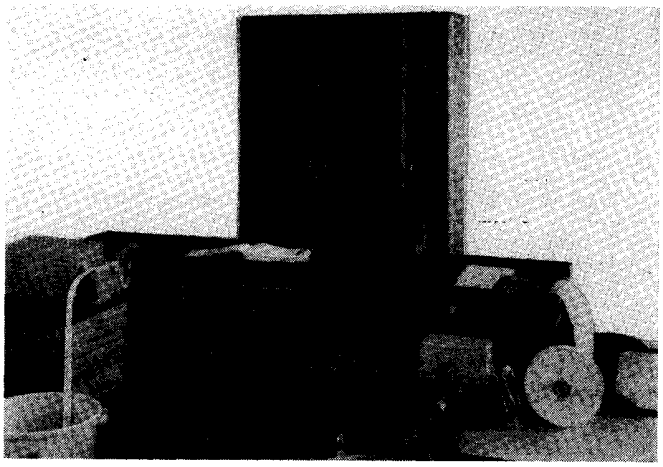
The *word length* is 42 bits and generally the cells are used for storing full-word or half-word instructions, and fixed-point or floating-point numbers, as shown in Fig. 1 (next page).

B. Central Unit

The central unit contains a lot of registers a.o. for handling the automatic address modifications, but for the user only the registers listed in the diagram in Fig. 2 are of interest.

C. Peripheral Units

The standard equipment comprises an 8-channel paper tape reader reading 1000 characters per second, an 8-channel paper tape punch punching 150 characters per second, and a typewriter which is used for both input and output.



The GIER computer.

Among the basic operations there is only one input operation, reading one character at a time; correspondingly there is only one output operation, printing or punching one character at a time. The wanted peripheral unit (or units) is selected by a previous operation. See also the list of operations, Fig. 3.

* Received September 9, 1963.

† Danish Institute of Computing Machinery, Danish Academy of Technical Sciences, Copenhagen, Denmark.

‡ Engineering Research Foundation, Division of Automatic Control, Technical University of Norway, Trondheim, Norway.

§ University of Copenhagen, Copenhagen, Denmark. Consultant to Danish Institute of Computing Machinery. Danish Academy of Technical Sciences, Copenhagen, Denmark.

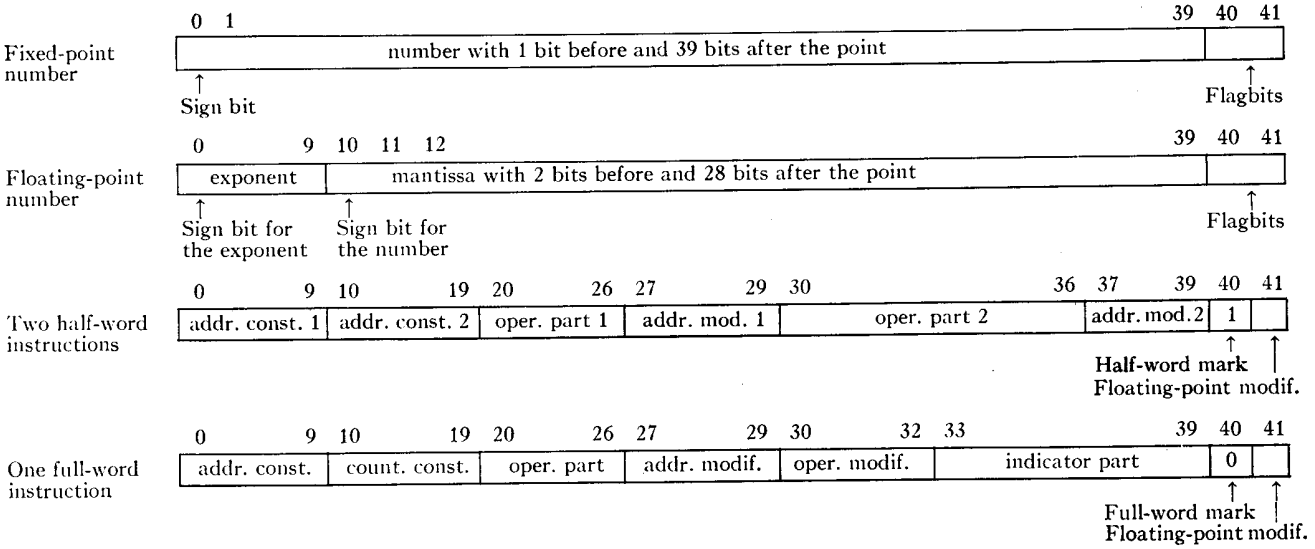


Fig. 1—Utilization of a cell. The 42 bits are numbered from 0 to 41.

Name	Number of Bits	Function
Accumulator R	41	Keeps the result of all 4 basic arithmetical fixed-point operations. The result is kept with 2 bits before the point. Shift and normalization takes place here. Together with the multiplier register used as one long register for double precision multiplication and division. In floating-point mode the accumulator together with 10 bits of the multiplier register form the floating-point accumulator.
Multiplier register M	40	Keeps the multiplier during fixed-point multiplication and the remainder after fixed-point division.
Index register p	10	The usual function of an index register.
Subroutine register s	10	Keeps the location of the last subroutine jump.
Indicator register	12	Keeps information about overflow, sign, zero-situation, and marking.
Track register	10	Keeps the address of the selected drum track.
Peripheral unit register	10	Keeps the numbers of the selected peripheral units.
Marking bits of accumulator	2	Keeps the marking bits of the latest used operand.
Overflow register	1	Keeps information about overflow for the latest arithmetical operation.

Fig. 2—The name, size, and function for the registers of interest to the programmer.

LIST OF OPERATIONS IN GIER

R is the accumulator. M is the multiplier register. (f) means that the operation can be floating-point modified.

Addition

ar(f) add to R.
an(f) add abs. value to R.
ac add R to cell.

Subtraction

sr(f) subtract from R.
sn(f) subtract abs. value from R.
sc subtract R from cell.

Multiplication

mk(f) multiply M and cell and add to R.
ml multiply. Result in long register.
mt multiply R by sign of cell.

Division

dk(f) divide R by cell.
dl divide long reg. by cell.

Shift

tk(f) shift in R.
tl shift in long register.
ck cyclic shift in R.
cl cyclic shift in long register
nk(f) normalize in R.
nl normalize in long register.

Boolean operations

ab add Boolean.
mb multiply Boolean.

Register and address setting

pm cell to M.
pp address to index register.
ps address to subroutine register.
pi address to indicator. Pattern of counting const. is a mask.
pa set count. const. as addr. const. in cell.
pc set count. const. as count. const. in cell.

Storing

gr(f) store R in cell.
gm store M in cell.
ga store addr. const. of R in cell.
gt store count. const. of R in cell.
gi store indicator in cell.
gk store track register and peripheral unit register in cell.

Jump

hv jump to left half, word (or full, word).
hh jump to right half, word (or full, word).
hs jump to subroutine. Store subroutine register. Set location of hs in subroutine register.
hr return jump. Restore subroutine reg.

Conditional operations

bs *if* address $\leq$ count. const. *then* skip next instr. (address does *not* include count. const.)
bt *if* address $\leq$ count. const. *then* skip next instr.
ca *if* addr. of R $\neq$ addr. of cell *then* skip next instr.
nc *if* addr. of R = addr. of cell *then* skip next instr.
cm *if* R = cell *then* skip next instr. Pattern of M is a mask.

Pseudo operations

it use addr. as count. const. in next instr.
nt use negative addr. as count. const. in next instr.
is use addr. as s-value in next instr.
ns use neg. addr. as s value in next instr.

Drum transfer

vk select drum track.
sk write on track.
lk read from track.

Peripheral unit functions

vy select peripheral unit(s). Pattern of count. const. is a mask.
sy write address as one character.
ly read one character to address part of R and cell.

Auxiliary operations

xr exchange R and M.
zq stop.
qq blind operation.
ud execute instruction in cell pointed out by the address.

MODIFICATIONS

Modification of basic operation

n clear R before basic operation.
f operate in floating-point mode (only 9 basic operations).
X exchange R and M after basic operation.
V skip next full word.
D direct modification: address is taken as an operand.

Indicator operations

I store overflow, zero situation, sign, or flag bits in indicator.
N *if* indicator condition fulfilled *then* skip basic operation.
L *if* indicator condition not fulfilled *then* skip basic operation.
M set flagbits in cell.

Fig. 3—Operations and modifications of operations.

D. Storage of Numbers

GIER is built with two different modes of operation in mind, namely fixed-point and floating-point mode.

Of the 42 bits of each cell the last two are used as flagbits; this means that they do not take part in the arithmetical operations but serve only administrative purposes. The remaining 40 bits are used for storing a number as follows:

A *fixed-point number* has 1 digit before and 39 digits after the point. Since negative numbers are represented by their 2-complements, the fixed-point range is the interval $-1 \leq x < 1$, and the sign of the number is shown by the first bit, this being 1 for negative numbers and 0 otherwise.

A *floating-point number* has a 10-bits exponent part, thus allowing exponents from -512 to 511. The remaining 30 bits constitute the mantissa which is normalized to one of the intervals $-2 \leq x < -1$ or $1 \leq x < 2$. It has thus 2 digits before and 28 digits after the point.

E. Instructions

Each cell can contain one full-word instruction or two half-word instructions. The structure of these two types of instructions is described in the sequel.

Half-word instruction: A half-word instruction occupies 20 bits of a cell, either pos. 0–9 and 20–29 or pos. 10–19 and 30–39, and consists of an operaton part and an address part. The flag bit in pos. 40 shows whether the cell contains one full-word or two half-word instructions. The flagbit in pos. 41 is used to distinguish fixed-point and floating-point mode for the arithmetic operations. (Thus always both of the two half-word instructions in one cell operate in the same mode.)

Operation part: The basic operation part of the instruction (in the external language written as two letters) occupies pos. 20–25 or pos. 30–35 allowing for 64 different combination of which only 61 are utilized so far. One bit (pos. 26 or 36) is used for the indication of clearing the accumulator before performance of the operation.

Address part: The remaining bits (pos, 0–9 and 27–29 or pos. 10–19 and 37–39) contain the address part including several possibilities for address modification. If none of the modifications are used the *address constant*, an integer between 0 and 1023 stored in pos. 0–9 or 10–19, is the address of the instruction in the usual sense. Pos. 28–29 or 38–39 are used to indicate one of the following three modifications.

r modification: The final address is the sum mod 1024 of the address constant and the contents of the control counter, *i.e.*, the address is calculated relative to the location of the instruction.

p modification: The final address is the sum mod 1024 of the address constant and the contents of the index register, *i.e.*, an index-modified address in the usual sense. GIER has only one ordinary index register, but in each cell pos. 0–9 may be used as an index register. (See below under full-word instruction.)

s modification: The final address is the sum mod 1024 of the address constant and the contents of the subroutine register. When a jump to a subroutine is performed the address of the jump instruction is stored in this register, and s-modified addresses are thus used in subroutines for communication with the main program especially for the return jump (and it is possible to use an arbitrary number of levels of subroutines, see below). If the subroutine register is not utilized this way it can act like an ordinary index register.

The last bit (No. 27 or 37) indicates *indirect addressing*. This means that the final address is calculated from the address part of that cell which the address of the instruction points out. Here the address of the instruction means the address calculated as above included r, p, or s modificaton. If the address part of the cell referred to is also indirectly modified, this address is again referring to a cell from which the final address is taken. There is no limit on the number of links in such a chain of indirect addresses.

Indirect addressing is often useful when a stored quantity is used many times. The address of the quantity is calculated once, and all later references to that quantity are made by means of indirect addressing.

Full-word instruction: A full-word instruction occupies a whole cell and consists of an operation part, an address part, a counting contant and an indicator part.

The flag bits in pos. 40–41 are used as above to distinguish between full-word and half-word instructions and between fixed-point and floating-point operations.

Operation part: Besides pos. 20–26 which are utilized exactly as for a half-word instruction the operation part of a full-word instruction comprises pos. 30–32. The contents of these bits indicate three independent modifications of the operation:

1) Exchange modification which means that the contents of the accumulator and the multiplier register are exchanged after the performance of the basic operation. This modification is often useful in connection with multiplications (where the multiplier has to be placed in the multiplier register) and when using the multiplier register as a working location.

2) Skip modification which means that the cell after the instruction is skipped and GIER continues with the instruction (instructions) in the second cell after the cell with the skip modified instruction. This device may be utilized when a single cell in a program is wanted as a working store, but is more important when used in connection with conditional instructions because it is very easy to make small local branchings in a program without proper jump instructions.

3) Direct modification which for the arithmetical operations means that the final address itself is used as the operand instead of referring to the cell where the operand is found. This modification is mainly used in connection with the address calculations and address administration of a program. Used in a storing operation the direct modification means that the storing takes

place in pos. 0–9 of the cell which contains the instruction, and hence it may change the address constant of the instruction. (If the address of the storing instruction is indirect modified the storing takes place in that cell which contains the final address part, see above: indirect addressing.)

The *address part* has the same structure and significance as in a half-word instruction except for the influence of the counting constant.

The *counting constant* is an integer between 0 and 1023 and it is stored in pos. 10–19. In most of the operations it has the following two effects: a) The final address is the sum of the counting constant and the amount from the address part (calculated as above). b) The address constant of the instruction is increased with the counting constant. This means that the address constant acts nearly as an index register; each time the instruction is performed the final address is automatically increased with the same number, namely the counting constant.

If the address part of the instruction is indirectly modified the counting constant is added to the address constant of that instruction from which the final address is calculated.

In a few administrative operations the counting constant has a different effect. For details see reference [1].

Indicator part: The indicator part of an instruction may be thought of as a subsidiary operation or a modification of the basic operation. There are three types of indicator operations but in each instruction there is only room for one indicator part. The three types are the following:

1) An *indicator-setting operation* which can be used in connection with any arithmetic operation—and in certain cases other operations too—for storing information in the indicator register. This *indicator register* contains 12 bits of which two named KA and KB can be set only from the control panel. The remaining bits are used for storing certain information about the result of arithmetical operations, namely, occurrence of overflow, the sign of the result, whether the result is zero or not, or the flagbits of the operand. Each basic instruction can have only one indicator operation and can store only one of the informations mentioned above.

2) A *mark-storing operaton* which can be used in connection with any of the normal storing operations for setting the flagbits of a cell. (A normal storing operation does not affect the flagbits of the cell.) The marking thus set may depend on the contents of the indicator register.

3) A *conditionalizing operation* which can be used in connection with *any* instruction to make it conditional. By means of this type of indicator part, the performance of any instruction can depend on the temporary state of the accumulator or on the contents of the indicator register. In the first case it may depend on the sign, the overflow, the zero situation in the accumulator, or on the flagbits of the last used operand. In the second case it may depend on the same sorts of information stored in the indicator register at an earlier stage of the program, or it may depend on the contents of the two bits KA and KB. KA and KB are thus two operator-controlled sense switches in the usual sense.

If an instruction has a conditionalizing indicator part, GIER starts by examining the relevant condition. If it is fulfilled, the instruction is performed; otherwise, the instruction is skipped and GIER continues with the following instruction.

III. List of Operations

A. Comments to the List

Fig. 3 contains a list of all the basic operations, the possible modifications, and the indicator operations. The notaton used is that of the input language SLIP (see Section IV-B). In Section III-C some examples of programming are found, and here only a few comments on some of the peculiarities will be made.

In *fixed-point* mode all arithmetical operations are carried out with two bits before the point, and hence it is possible to operate in the interval $-2 \leq t < 2$, but only to store numbers between -1 and $+1$.

It is possible both to add and subtract directly in storage (the operations ac and sc).

The multiplication operations mk, ml can perform *accumulating multiplication.* For the operations ml, dl, nl, tl, cl the accumulator R and the multiplier register M act as one long register with two digits before and 78 digits after the point (the register is denoted RM).

The shift operations tk, tl can shift both right and left. The normalization operations nk, nl, which normally shift to the left, may in case of overflow shift to the right, thus facilitating the handling of results with overflow.

The floating-point version of tk and nk are especially designed for the *conversion of numbers* between fixed-point and floating-point mode.

In the operation pi (store in indicator register) the binary pattern of the *counting constant* in the instruction *acts as a mask* which may keep some bits in the indicator unaltered. This is very useful because the indicator often is used as 10 independent 1-bit registers. The same masking effect occurs for the vy operation, because in the external unit register also, the bits may be used independently for selecting different external units.

There is no conditional jump in the classical sense because each and every instruction can be conditional by means of an indicator part.

Correct use of the *jump instructions hs and hr* makes it easy to program with arbitrary many levels of subroutines: The hs jump to a subroutine stores the old contents of the subroutine register and sets its own location in the register; hence the instruction hr s+1 performs a return jump to the instruction just after the hs jump, and furthermore the subroutine register is restored.

The five *conditional operations* share the following branching feature: Used in a left half-word instruction only the following right half-word can be skipped. Used in a right half-word or in a full-word instruction the following full-word can be skipped, whether the contents of this are one or two instructions.

The *pseudo operations* it, nt, is, ns act always on the immediately following instruction. They are rather special operations and are a.o. used when packing a program in a minimum of storage is wanted.

Drum transfer by the operations sk and lk may go on simultaneously with other operations but is always completed before a new vk, sk, or lk instruction is performed.

Each of the indicator operations must be supplemented with an *indicator address* showing the wanted condition or the wanted part of the indicator register.

B. Operation Times

When speaking of operation times in GIER, two characteristics must be taken into consideration: First, the time spent on the address calculation depends on how complicated the actual address part is, and secondly, some of the basic operations take different time under numerically different circumstances.

Therefore, the table in Fig. 4 gives mean operation times for the basic operations, included the time (27 μsec) for the inevitable basic address calculation, this comprising r-, s-, and p-modified addresses. The operation times for the remaining possible modifications are found in the little table in Fig. 5 which also shows the sequence of performance of the different parts of an instruction.

Operation	Time in μsec	
	Fixed-point	Floating-point
qq-ps-pp-vk	29	—
gr-gm-ga-gt-gp-gs-gi-gk pm-pa-pt bs-bt-xr lk-sk-hv-hh	36	36 (only gr)
it-nt-is-ns pi-mb-hs-vy	33	—
ar-an-ac-sr-sn-sc mt-ca-nc-hr	49	100
ab-cm	56	—
tk-tl-ck-cl-nk-nl	60	70
mk-ml	180	165
dk-dl	270	220

Fig. 4—Operation times for the basic operations, including 27 μsec for the basic address calculation.

Sequence of performance	Time in μsec
1) Indicator condition checking	
a) if condition not fulfilled	15
b) if condition fulfilled	0
2) Address calculation (the basic time of 27 μsec is included in Fig. 4)	
a) if counting constant≠0, then add	9
b) if indirect address but no s modification, then add for each link of the chain	12
c) if indirect address and s modification, then add for each link of the chain	26
3) Basic operation	see Fig. 4
4) Indicator setting, mark storing, V modification	0
5) Exchange modification	4

Fig. 5—Sequence of performance of the different parts of an instruction and the corresponding operation times.

C. Examples

Below, a few examples of computational techniques in GIER are shown, and in Section VI-D some more examples of programming may be found.

Example 1: To show some of the features of the indicator part we shall consider a little problem: A set of numbers are stored in cell nos. 100, 101. · · · Some of them are A-marked and the last of them has a B marking. We want to store the sum of all the A-marked numbers in cell no. 500. In the notation of the SLIP language (see the following section) a piece of a program for this problem may look like this:

	oper. part	addr. part	count. const.	indic. part	
[1]	grn	500			clear cell number 500
[2]	arn	99	t1	IPC	take cell number 100+i to the accumulator; put the flagbits in the indicator register
[3]	ac	500		LPA	*if* there was an A marking, *then* add the accumulator to cell number 500 *else* go on
[4]	hv	r−2		NPB	*if* there was no B marking, *then* jump back and take the next number *else* go on

After the performance of this program the address part of instruction no. 2 is changed and contains the address of the last number in the set.

Example 2—Scalar product: Let us take the very frequently occurring process of forming the scalar product of two vectors. Given the dimension of the vectors we may carry out this process by the ordinary use of an index register. But GIER allows a very convenient and rapid calculation as follows.

Let the two vectors each be stored in consecutive cells, starting in no. a+1 and b+1, respectively, and let the last element of each vector be A-marked. The scalar product may then be calculated by three instructions

pm	a	t1		take element from cell a+i to M
mk	b	t1		multiply M by cell b+i and add to R
hv	r−2		NA	*if* no A mark *then* jump back *else* go on

assuming that the accumulator is empty when starting. No information concerning the dimension of the vectors is necessary.

Example 3—Double precision arithmetic: The way in which the arithmetic of the machine has been constructed permits a very easy handling of double precision numbers.

A double precision number will occupy two cells which are normally consecutive. If the contents of the two cells are denoted a1 and a2, a1 being the most significant part (the head) and a2 the least significant part (the tail), the number in question has the value a1 + e×a2, where e is the number 2^{-39}. Due to the fact that the machine is a binary machine representing negative numbers by their 2-complement, the tail may always be taken positive, a fact which leads to a convenient representation. If, *e.g.*, a = a1 + e×a2 is the negative number

```
1.11010011 . . . 1110011  11001010 . . . 011101
↑                 ↑                      ↑
pos 0             pos 39                 pos 78
```

then a1 is the negative number 1.11010011 · · · 110011 and a2 the positive number 0.11001010 · · · 011101.

A double precision number may be kept in the long register RM. Operations involving the long register RM are ml, dl, tl, nl, and cl, and are called long operations. In these operations the position 0 of the M register does not take part.

Let now b = b1 + e×b2 be another double precision number held in the two cells no. 100 and 101. *Addition* of b to the number a in the RM register may be executed by the instructions

[1] gr 99 X	store the head of a in a working location, cell no. 99, and interchange the contents of R and M
[2] ar 101, tl−39	add a2 and b2 and put the result in M by means of a long shift. If the sum was >1, R now contains a one in pos. 39, otherwise R contains zero
[3] ar 100, ar 99	add a1 and b1 to R thus forming the head of the result

Changing the operation part into sr in the instructions ar 101 and ar 100, we get a double precision *subtraction*. Here the instructions in [2] will leave R with 0 in all positions, if the result of the left half-word instruction is positive, and otherwise with ones in all positions. The tail of the double precision result is still positive.

The *product* of the two numbers is, except for a maximum error of one unit in the 78th position,

$$a \times b = a1 \times b1 + (a1 \times b2 + a2 \times b1) \times e.$$

The accumulating properties of the short multiplication (mk) and the long multiplication (ml) make possible the following piece of program for the double precision multiplication under the same initial conditions as before.

[1] gr 99, mkn 100	store the head of a, place the product a2×b1 in R, rounded off to 39 bits
[2] pm 99, mk 101	take a1 to M, multiply a1 by b2 and add it to R, *i.e.*, form the sum a1×b2 + a2×b1 in R
[3] ml 100	multiply a1 by a2 (long multiplication) and add the contents of R to the lower part of the product thus forming a×b in one sweep

It will be seen that ml in fact treats the contents of R and M as integers having their unit positions in pos. 39 of R and M.

As it is to be expected, the double precision *division* is somewhat more complicated.

Still assuming the same initial conditions we suppose furthermore that a and b have been normalized. This will give maximum accuracy. Let c = a1/b1 to 39 positions. Then the quotient q is, except for a maximum error of one unit in the 78th position,

$$a/b = q1 + e \times q2$$
$$= c + ((a1 + e \times a2) - c \times (b1 + e \times b2))/b1.$$

The division may be performed by the following instructions:

[1] gr 99, gm 98	store a in two working locations
[2] dl 100, gr 97	the quotient c = a1/b1 is stored in a working location
[3] mtD−1 X	reverse the sign of c and take it to M
[4] mkn 101, ar 98	form a2 − c×b2 in R
[5] ml 100, ar 99	form a1 − c×b1 + e×(a2 − c×b2) in the long register RM
[6] dl 100, ar 97	form c + RM/b1 in R. The remainder from the division is found in M
[7] gr 99 X	store q1 and take the remainder to R
[8] dk 100 X	the division always gives the least absolute remainder having the same sign as the divisor. Therefore [8] yields the positive tail q2 and places it in M
[9] arn 99	take q1 to R so that RM now contains the quotient

The foregoing programs may easily be written as closed subroutines, but, except for the division, not much will be gained.

IV. Help and Slip

A. The Administrative System HELP

GIER is equipped with an administrative system called HELP for facilitating testing and running of programs. The system comprises an interrupt mechanism, activated through the HELP button, a central administration program monitored by typewriter input, and a number of subroutines among which the input program SLIP is the biggest and most important. The others are subroutines for normal output, for control output, for initializing, for comparison of storage sections, and for supervising a program during the run.

In this section we pass in review the main features of the central administration and some of the subroutines, while SLIP is treated in the next section.

HELP button and central administration: Since the ferrite core memory is rather small the system is designed so that it occupies only 10 cells of the ferrite core store during the run of a program. On the other hand it is obvious that during an interrupt the system must have access to a much larger part of the core store and yet be able to restore the total store before the running is continued.

This is obtained by reserving the last 26 tracks of the drum for an *image* of the ferrite core store during the interrupt. Since the system itself occupies the first 58 tracks of the drum the *available store* for the programmer consists of 1014 cells of the core store and about $\frac{3}{4}$ of the 320 drum tracks.

Of the reserved 58 tracks the first 32 are locked for writing so that it is impossible during a normal run to destroy the fundamental part of the HELP system.

At any stage of a run one can call for an *interrupt* by pressing the HELP button which has the following effect: The contents of the registers and the core store are stored in the image on the last 26 drum tracks, and control is transferred to the central administration which then is waiting for typewriter input telling what action is wanted. It is now possible to start any of the HELP subroutines or to make corrections in the stored program.

When the desired encroachment has been performed, an end signal must be typed. Then the core store and the registers are restored from the image and control is transferred to the point of the program where it was interrupted.

HELP subroutines: We shall divide the subroutines roughly into three classes according to their use before, during, and after a run:

1) Before a run is begun one may use a subroutine for initializing the whole computer. After input of the program this can be copied to an unused part of the drum for later comparisons or for restoring the initial situation if something goes wrong during the run.

2) During the run one may use subroutines supervising the program, *i.e.*, the jumps performed, or the numerical behavior, for instance, by typing out all changes of a chosen register or cell. One may invoke control output at every performance of a selected instruction in a program loop. HELP also contains subroutines for the normal output of text and numbers.

3) After the run subroutines may be used for control output of any part of the store, for comparison between the program before and after the run, and for output of the corrected program in a compressed form suitable for fast input.

If a user wants additional facilities it is easy to enlarge the HELP system to include new subroutines either in addition to or instead of some of the standard routines. If, on the other hand, the maximum available storage is wanted, it is possible to confine the HELP system to 26+39 drum tracks instead of 26+58 tracks at the sacrifice of some of the subroutines. But the 26+39 tracks are necessary if the interrupt mechanism, the central administration and the input rotuine SLIP shall be intact.

B. The Input Routine SLIP

The coding language used on GIER is called the SLIP language, *i.e.*, the language accepted by the input routine SLIP (which means *s*ymbolic *l*anguage *i*nput *p*rogram). SLIP reads *instructions*, *textstrings*, and three types of *numbers* namely fixed-point and floating-point numbers and integers (which may be packed with at most four integers in each cell). It is allowed to include comments (for instance in square brackets) in the input string, and they are skipped by SLIP.

SLIP is a subroutine in HELP and is always invoked through the central administration of HELP; hence all input which is meant to land in the ferrite core memory in fact is put, by SLIP, into the *image* on the drum and not till the end of the input process placed in the ferrite core store.

An important feature of SLIP is that in instructions *symbolic addressing* is allowed. This means that the address constant and the counting constant of an instruction may be *symbolic names* whose values are defined through the use of the names as *labels* elsewhere in the program (only a rather restricted class of names is available). With respect to the use and scope of such names the SLIP language has a block structure very much like that of ALGOL: 1) Names must be declared in a block head before use and can only be used within the block with the declaration, *i.e.*, they are *local* to that block. 2) If a name is declared in each of several blocks inside each other, the name may have different values on each block level.

Special regard is taken to make it easy to use symbolic names together with the relative addressing.

During input an extensive *syntactical check* is performed and whenever an error is found a message about it is typed out. Then SLIP continues the reading, skipping the erroneous instruction or number. This implies that often all the syntactical errors are found in one sweep, and it also implies that if there is only one or a few benign errors in a program, these may be corrected on the spot by means of HELP, and a test run can be carried through in spite of the errors.

V. Structure of the ALGOL System

A. The Background and Aims

The final decision that an ALGOL compiler should be written for GIER was made in January, 1962. This decision was based on a significant amount of previous local experience. An ALGOL compiler for the machine DASK had been developed during the years 1959 to 1961 and during its actual use had proved to be a tremendous gain in the utility of that machine. Since DASK is rather similar to GIER as far as storage capacity and speed is concerned the great value of having an ALGOL compiler on the GIER was therefore obvious.

At the same time, the success of DASK ALGOL with its users had not made its designers blind to its shortcomings. In fact, already during the later phases of the development of DASK ALGOL it had become increasingly clear that it was poorly designed in many respects and the systems programming group was quite keen to have a new try, using the accumulated experience. There was therefore a very happy match of supply and demand.

The following paragraphs review the highlights of the principles underlying the GIER ALGOL system. For a much more detailed report, which also includes references, see reference [3].

In designing GIER ALGOL we tried to develop a practically effective programming system, based on the generality of notation inherent in ALGOL 60. More particularly, the compiler should be fast and should include extensive error checking facilities, and the system should to a large extent relieve the user of having to think of the two stores, cores and drum. The language should include a generous helping of ALGOL 60, avoiding minor restrictions as far as possible. Thus there should be no limit to the number of characters used to identify the quantities, and the powerful procedure facilities, including recursive uses, should be included. Essentially, only one part of ALGOL 60, the so-called own arrays, were excluded.

The background for these design goals was on the one hand a belief in their value for the programmer, and on the other hand the conviction that a design along these lines, if pursued consistently, would entail no essential compromises.

Our previous work had indicated conclusively that in designing an ALGOL system it is essential to start by solving the problems of the execution of the finally translated program. The fact is that ALGOL 60 contains certain basic elements, *e.g.*, the block structure and procedure calls, which require administrative action at execution time not corresponding to actions which are built into the present-day machines. In addition, administrative actions must be included in a system which will take care of the transfer of information between the core and drum stores during program execution. Actions of this nature are taken care of by what we call the *running system.* The design of the total system therefore comprises two major parts, the *running system* and the *translator*, to be attacked in this order.

The next level of design of GIER ALGOL, whether the running system or the translator, is based on storage allocation considerations, dictated by the inhomogeneity of the store of the machine. A poor design in this respect leads to a slow and painful writing of the system, to slow compilation, and to inconvenience for the user, as experienced with DASK ALGOL.

B. Running System, Storage of Variables

The problem of storage allocation during program execution has two parts: storage of variables and storage of instructions. By January, 1962, it was abundantly clear that the proper way to store the variables of an ALGOL program is to use a part of the fast store as a stack. By this method the block structure of the ALGOL program can be used fully for economizing the demand on storage and the generality of ALGOL 60 with respect to arrays having dynamically changing sizes and procedures calling each other and themselves in arbitrary ways is handled without difficult.

As an illustration consider the following skeleton of a program. (The dots, · · ·, indicate some statements doing the useful work of the program.)

```
begin integer n;
for n:=5, 20 do
  begin integer k; array A[−n:n];
  . . .
  for k:=6, 8 do
    begin array B[0:n, 1:k];
    . . .
    end for k;
  . . .
  end for n;
. . .
end the program.
```

The meaning of this program with respect to the dynamic existence of variables is the following: In the outermost level of blocks only the variable n exists. On encountering the *for* statement we are supposed to execute the controlled statement starting with

```
begin integer k;
```

and ending with

```
end for n;
```

twice, putting n equal to, first 5 and then 20. The controlled statement in this case calls for the establishment of a simple variable k and, more interesting, in an array A having the first time 11 elements, A[−5], A[−4], · · · A[4], A[5] and the second time 41 elements, A[−20], A[−19], · · · A[19], A[20]. Now while these two versions of the controlled statement are being executed we are further supposed to execute the other *for* statement, which again calls for two executions of its controlled statement. This means that this inner statement will be executed 4 times with the following number of elements in the two arrays A and B:

n	k	Number of elements	
		in A	in B
5	6	11	36
5	8	11	48
20	6	41	126
20	8	41	168

The problem of accomodating these arrays economically in a linear store might seem to be a rather nasty problem. However ALGOL 60 is very helpful because the establishment of arrays (and indeed the introduction of variables) is always associated with the nested blocks. Specifically, in the above illustration the array A will never be established or deleted at a time when the array B exists. This is the reason why a stack (or push-down list) is such a convenient way of arranging the storage of variables in ALGOL 60. In GIER ALGOL the stack uses the locations from about location 825 and downwards towards smaller addresses. If we follow the execution of the illustrative program step by step as entries into and exits from the blocks are made we get the following addresses of the variables. (The arrangement is somewhat simplified in order not to burden the reader with too much detail.)

After entry into program	n:825
After n:=5	k:813, A:814–824, n:825
After k:=6 with n=5	B:777–812, k:813, A:814–824, n:825
After k:=8 with n=5	B:765–812, k:813, A:814–824, n:825
After inner for statement, n=5	k:813, A:814–824, n:825
After n:=20	k:783, A:784–824, n:825
After k:=6 with n=20	B:657–782, k:783, A:784–824, n:825
After k:=8 with n=20	B:615–782, k:783, A:784–824, n:825
After inner for statement, n=20	k:783, A:784–824, n:825
After outer for statement	n:825

Two conclusions emerge from this discussion: 1) By using a stack for variables the locations reserved for them will form a tight sequence having a length which varies during the execution of the program, but being fixed in position at the one end. 2) The storage allocation will be dynamic, *i.e.*, the addresses of the variables cannot be finally calculated until the program is executed.

In this form the system will only allow variables which are stored in the cores. The system will not automatically handle the storage of variables on the drum. Instead, there are available built-in procedures for transferring arrays of variables between the two stores. Thus, as far as variables are concerned, the programmer may regard the drum as an output/input medium. The conventions of the standard procedures for performing the transfers are such that the programmer is unable to refer to absolute locations on the drum. This is desirable in order to avoid a dependence between this use of the drum and its use for storing the program and the compiler.

C. Running System, Storage of Program

The running system includes a fully automatic administration of the transfers of program segments from the drum to the cores. Generally speaking, this administration tries to use all the available core store for those sections of the program which currently seem to be of most interest.

This is implemented as follows. Primarily the program is stored on the drum. The translator has segmented it into drum track sections, including a special instruction at the end of each segment. The program of each segment has such a form that it may be executed correctly from any position in the core store, without making any assumptions as to the presence of other program segments in the core store. In particular, jumps within the instructions of the segment use r-modified addresses (address relative to the current location of the instruction). On the other hand, jumps to points in the program stored in other segments are represented by a jump to a fixed administration routine in the running system and an additional description of the destination comprising the track number of the segment and a track relative address.

In executing the program the running system acts as a monitor. It has information on the current extent of the stack and will therefore be able to divide the remaining part of the core store into a certain number of program segment places, at least 2 and at most 20. At all times it keeps a table of the track number of the segment currently occupying each available segment place. Whenever the program jumps from one segment to another the monitor takes over, checks its table to see whether the desired segment is present in the core store, and if so jumps to the proper location within it. If the segment is not present, the monitor will transfer it from the drum to that segment place in the core store which for the longest time has been left unused. For this purpose the table of the available segment places contains a "cycle number of last usage" in addition to the track number.

D. Running System, the Administration Program

It is clear from the above that the execution of a translated ALGOL program requires the presence of certain administrative routines in the core store (dynamic addressing, segment monitor, etc.). These use about 200 words at the top of the store and in addition one extra word adjacent to each available segment place.

Some of the run time of a program will of course be spent in the running system. It is of interest to note that the significant part of this is related to the limitations of the machine and the translator, and not to the generality of the language which the system will process. In fact, in several realistic programs (realistic in the sense that they perform numerically useful processes such as inversion of matrices, while forgetting about exotic features of ALGOL 60) the major bottlenecks, in execution time, were 1) subscription of variables and 2) transfer of control between segments already present in the core store. Both of these processes might be made relatively insignificant if a few special instructions were incorporated in the machine.

E. The Translator, Storage Problems

During translation the machine is dealing with three different bodies of information: the program of the translator itself, the text which is in the process of being translated, and tables of descriptions of the objects of the source text. If this information is to be handled efficiently in a machine like GIER, it is essential that random, or almost random, references to information kept on the drum be avoided. This suggests the following approach: First, in order to avoid jumping about in the program of the translator the translation should be divided into a sequence of separate processes, each so simple that its logic can be held in the cores. Second, the text should be processed by means of sequential passes, scanning the source text or intermediate versions of it from one end to the other. These two first points indicate that a multipass translator should be used. Third, the translator should be constructed in such a manner that all necessary tables can be held in the core store while they are used. This in conjunction with the demand on the division of the translator program shows that during certain passes the cores should

be free to hold large tables, while the logic of the pass should be very simple.

A closer analysis of this approach shows that by using about 10 translation passes all the above requirements will be satisfied at the same time as the translation speed will be very satisfactory. The final GIER ALGOL system in fact uses 9 passes and produces about 30 final machine instructions per second.

In the standard version of the compiler the translator program occupies tracks 66 to 176, while tracks 39 to 65 hold the run-time routines (the running system, standard procedures, etc.). Tracks 0 to 38 are then free to hold the HELP complex. The translator uses tracks 177 to 319 for holding the partially translated program. Since these tracks are used in a cyclic manner they are all available for the finally translated program.

One great advantage of the multipass method is that a suitable alternation between forward and backward passes is possible. This takes care of internal references in a most convenient manner. Two out of the 9 passes in GIER ALGOL are backward passes.

F. General Administration of Translator Passes

In a multipass translator the question of the intermediate languages used to communicate the partially translated text from one pass to the next becomes prominent. In GIER ALGOL the solution adopted for this was strongly influenced by considerations of the check-out of the translator. Clearly, during check-out the output produced by each pass must be made available for inspection. Unless this is taken care of in a general manner the print-out of this information may turn out to be a significant source of trouble and inconvenience in writing the translator.

For these reasons the following scheme was adopted: All intermediate languages are expressed in terms of uniform strings of 10-bit symbols called *bytes*. In other words, the output from a pass will consist of a series of integers in the range from 0 to 1023. The packing of these symbols with 4 in a word and 40 words in a drum track and the transfer of this output to the drum, and the analogous actions on the input side, may then be performed by a general pass administration which is the same for all passes. For check-out purposes this general administration is extended with a facility for printing the decimal values of the symbols as they are received for output from the pass.

By further testing the passes in strict order, starting by pass 1, the test data used for checking the translator logic will all be written in the ALGOL source language, although of course the test programs must be written specially for each pass to make sure that the logic of the pass is adequately covered. This method proved to be extremely effective and contributed greatly to the early completion of the system. In fact, more than half of the system was loaded into the machine for the first time and checked out during the period August 2 to 24, 1962, after which time the system could be distributed to all GIER installations.

G. Translation Passes

The actions of the 9 translation passes are as follows:

Pass 1. Conversion of the hardware representation of symbols to a string of reference language symbols of ALGOL 60.
Pass 2. Matching of the free identifiers.
Pass 3. Analysis and check of the delimiter structure of the text.
Pass 4. Collection of the declarations of the identifiers of the program into tables.
Pass 5. Distribution of the descriptions of the kinds, types, and storage, of quantities to all the places in the program text where the identifiers occur.
Pass 6. Check of consistency of text with respect to kinds and types of quantities.
Pass 7. Conversion of expressions into a sequence of machine operations.
Pass 8. Segmentation into drum track segments. Establishment of internal jump references.
Pass 9. Sorting of the final segments on the drum.

VI. The Machine Code of the ALGOL Compiler

A. Comments

One of principal decisions in the design of the ALGOL compiler (see Section V-A) was that each pass should be allowed ample room in the core store for all the program and tables concerning it. This decision was partly inspired by our somewhat painful experiences from the DASK ALGOL, trying to fit a too big lump of program into a small core store.

This decision made the coding of the single passes a comparatively straight forward job once the algorithms were written. Even so it was clear that, to keep the number of passes down and to avoid dividing a pass in two when it logically ought to be one, it would be necessary to pack the information and to use, if not clever, then at least not too wasteful machine-coding techniques.

The efficiency of the machine program is closely connected with the representation of the information it has to handle. In GIER, as in most computers, the address is that unit of information which is most easily handled; it was therefore decided to quantitize the information between passes in bytes of 10 bits each, corresponding to the length of the address part of an instruction. A computer word can thus hold 4 bytes.

This scheme is strictly adhered to in the communication from pass to pass and is the preferred representation in the internal tables too, although these are treated more freely.

The following sections will discuss some of the operations involved in the compilation process and the GIER instructions performing them.

B. Operations in the Compiler

With the above in mind, the kinds of operations performed in the compiler may roughly be characterized as follows:

1) Handling of Booleans and conditions. Setting of Booleans, testing of Booleans and byte relations.

2) Subroutine calls.

3) Handling of bytes. Unpacking and packing computerwords, simple computations on byte values, table indexing and switching on byte values.

C. GIER Instructions and Their Use in the Compiler

1) *The handling of Booleans and conditions.* One of the characteristics of compiler operations is that many of the operations governed by a condition are very simple and often can be performed by one or two machine instructions. In the GIER any instruction can be conditioned. This means that such short conditional operations can be performed directly, instead of by use of conditional jumps, and will therefore result in a fast and compact machine code.

There are 3 ways in which an instruction can be conditioned: 1) The use of the indicator part of the operation can make the execution of the rest of the instruction dependent on the status of the accumulator, the R register; the tests which can be performed here are the usual ones: test on sign, on overflow or on zero. Furthermore the status of the flagbits of the R register, *i.e.*, of the last operand brought to the R register, may be tested. 2) In the same manner the status of a pair of indicator bits may be tested (see Sections II-E and III-A). 3) Finally the result of a test performed by one of the special address compairing instructions may cause the next instruction to be skipped. (See examples below of the use of the instructions ca, nc, bs and bt.)

The above means that the indicator and the flagbits of operands are very convenient for the storing of Booleans, as the setting and sensing of these often can be done as part of other instructions.

Another important feature is the V modification, which causes an unconditional skipping of the next word. This facilitates the coding of if . . . then . . . else conditions.

The extent to which some of the above facilities are used can be illustrated by the following counting of different instruction types in one of the actual translator passes (pass 8):

Number of conditionings by:

Indicator bits	37
Status of R register	12
Instructions ca and nc	3
Instructions bs and bt	11
Total:	63

Kinds of instructions conditioned:

Ordinary jumps	14
Subroutine calls and returns	20
Other operations	29
Total:	63

Instructions setting indicator bits or flagbits: 47

Total number of instructions in pass 8 is around 400.

In contrast to what might be expected, practically no use is made of the general Boolean operations ($\wedge$ and $\vee$) operating on groups of Booleans. This is a consequence of our general philosophy, that questions of the type, "Do you belong to this or this or this . . ." category, should be avoided. Where the logic requires such questions to be answered, the table description of the item in question will instead hold one Boolean for each such combined condition, thereby avoiding the rather empty chain of reactions: "It's not me, it's my colleague."

2) *Subroutine calls:* Another counting in the same pass showed 55 occurrences of subroutine calls.

The subroutines themselves are normally quite short (5–20 words). The main part of the instructions are common for two or more subroutines, which only differ in the first couple of instructions.

In many computers the handling of such nearly alike subroutines may be quite cumbersome and space consuming, because the reference back to the main program is stored at the entry point to the subroutine and therefore has to be moved to a common location in case of more than one entry point.

The subroutine mechanism in the GIER (see Section III-A) makes such use of subroutines easy, and it will still provide for the call of subroutines within subroutines because of the special feature which stores the previous content of the subroutine register (the s register) in the call instruction before performing the jump.

3) *The handling of bytes:* The instruction list is abundantly rich in instructions dealing with addresses. In effect it may often be a problem to choose among the multitude of possibilities. For unpacking and packing the normal arithmetic operations, shift operations, and operations for storing full words or address parts are available.

As may be expected when handling bytes, the most common number of shifts performed is 10. Again a counting of instructions in pass 8 will show this:

Instructions performing 10 short right or left shifts:	19.
Other shift instructions:	5.

Furthermore, the input-output mechanism of the central administration is constantly shifting bytes by multiples of 10. This may suggest that some fast instructions for these special shifts might be useful. (A shift of 10 takes around 70 μsec.)

Quite complex operations with bytes can be handled without use of the arithmetic operations; some of the possibilities are shown in the examples below.

D. Examples

The following examples will each consist of a small piece of machine code, an equivalent algorithm, and in some cases a comment, in three columns. For the function of the single operation codes see Section III-A. Half-word instructions may appear two in a line separated by comma, or they may stand by themselves. Full-word instructions will be characterized by a t separating the address part and counting part.

1) Simple byte manipulations.

```
1  ga 100 , gt 101      a:=address part of R; c1:=counting part of R;
2  pa 100 t 27          a:=27;
3  it (101) , pt 100    c:=a1;
4  is (101) , it s+3    a:=a1+3;
5  pa 100
6  pp (100)             p:=a;
```

The operand of the is instruction is used as value for s in the next instruction

2) Table indexing. Table starts at location 1000.

```
1  arn (100)t 1000 IPC  a:=a+1000; R:=content [a]; Indicator bits PA
                        and PB:=flagbits [a];
```

In the same operation the contents of a computer word are brought to the R register and the flagbits of the word are stored in the indicator for later tests

3) Conditioning on byte values.

```
1  ca (100) , pt 101    if a=address part of R then c1:=0;
2  can p−27 , it (100)  a1:=if p=27 then a else 0;
3  pa 101 ,
4  bs (100) , it (101)  a2:=if a>0 then a1 else 27;
5  pa 102 t 27
6  bs p+503, pp p+7     if p>−503∧p<9 then p:=p+7;
```

The letter n indicates a clearing of the R register

Addresses are treated with sign whereby addresses above 511 are taken as negative

4) Conditioning on status of R or indicator bits.

```
1  qqn (100)t 7 LT      if R is negative then begin R:=0; a:=a+7 end;
2  arD (100)t 34 NPA    if indicator bit PA=0 then begin a:=a+34; ad-
                        dress part of R:=address part of R+a; end;
3  it (100)t 37 LPC     if PA=1∧PB=1 then begin a:=a+37; c:=a end
4  ptn 100t 77          else c:=77; R:=0;
```

5) Compound conditions and more complex operations.

```
1  bs s+502 t 506       if s>4∧s<10 then a1:=a1+3
2  qqV (101) t 3
3  it (100) , pa 102    else a2:=a;
4  it (101) , is (100)  a:=a+a1;
5  bs s+502 t 506       if a>4∧a<10 then a1:=a1+3
6  qqV (101) t 3
7  it (100) , pa 102    else a2:=a;
8  bsV (100) t 57 NT    if R is negative then begin R:=0; a:=a+200;
9  hvn (100) t 200 IZA  ZA:=1 go to instr[a] end;
10 gp 100 , it (101)    if a>57 then begin a:=p; a2:=a2+a1 end else
11 qq (102) t 1         a2:=a2+1;
```

Compare with the example below

This shows the use of the is instruction for providing a temporary extra index register

This is an example of one conditional instruction leading to 3 different actions

To get as readable and uniform notation as possible, we will assume that the address parts of cells 100, 101, 102 hold the bytes a, a1, a2, respectively, while the corresponding counting parts hold the bytes c, c1 and c2. The contents of the index register and the subroutine register (which can be used as an ordinary index register) will be denoted by p and s, respectively.

The above examples, which all are slightly modified examples from the actual compiler, shows the powerful instruction list in the GIER.

It may be claimed that some of the examples are on the border of "trick coding" but why should the mechanisms not be used when they are available. The main problem in this connection is the documentation which of course has to be kept fully up-to-date.

VII. Structure of GIER (from the Technicians' Point of View)

At the planning of the hardware of the GIER computer it was realized that for several reasons the aim had to be toward the greatest possible flexibility in the structure. First, the order structure was not finally fixed at the time when the construction of the machine was started. Secondly, changes and expansions had to be expected. Consequently, it was decided that the communication between the registers should go on via a common busline system and that the microprograms should be housed in a changeable fixed wired store. The store size and type, 1024 words in a core store and 12,800 words in a drum, was mainly dictated from the economy. To gain speed parallel structure was chosen, and transfers to and from the drum should go on simultaneously with other operations. Furthermore, it was decided to run the circuits in synchronous mode, except for the drum, at a clock frequency of about 500 k-z.

A. Information Transfer System

In Fig. 6 is shown all the transfer lines in the machine. As mentioned all information transfer between the registers is going on via a common busline system consisting of 42 wires. (The word length is 42 bits.) Communication with the core store is parallel 42 bits at a time, while the drum is only accessible serially bit by bit, via the 42-bits buffer register T1. When a full-word is shifted through T1 to or from the drum the next word is passed to or from the core store via the buslines interlaced with normal operations. Transfers are released a whole track (40 words) at a time. Input and output from and to external units is transferred via the registers bl and bs one character at a time.

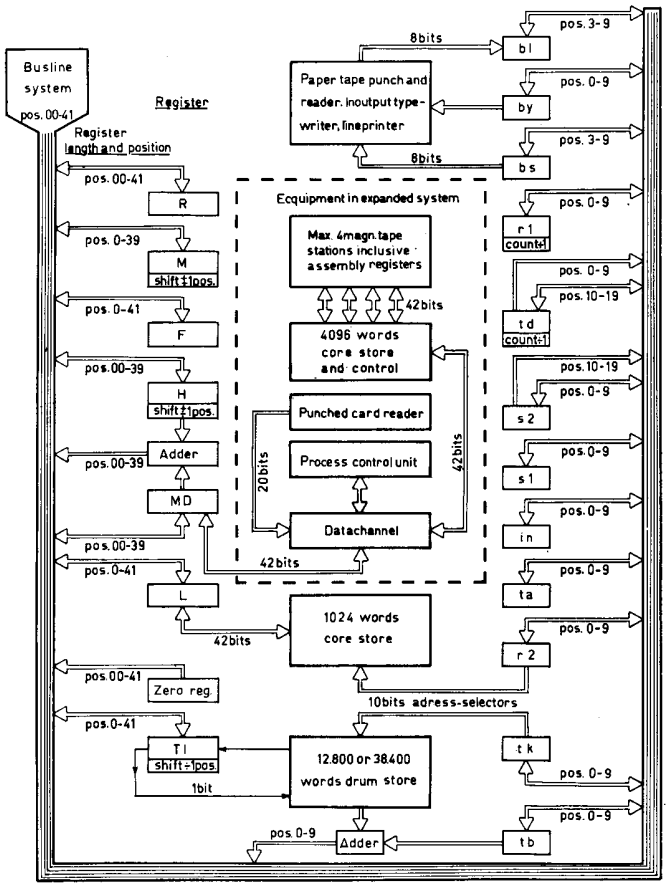


Fig. 6—Communication channels in GIER.

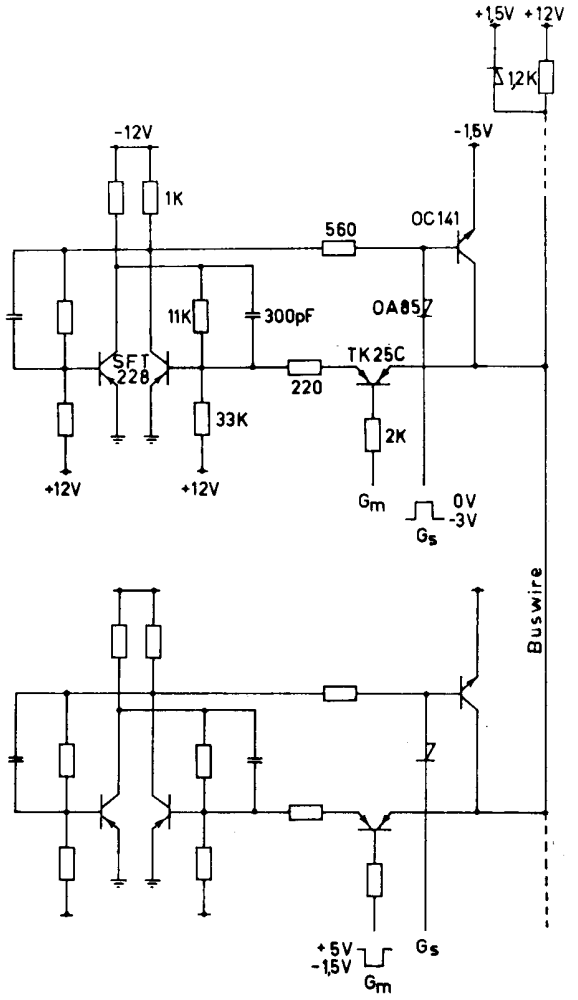


Fig. 7—Register elements with input and output gates.

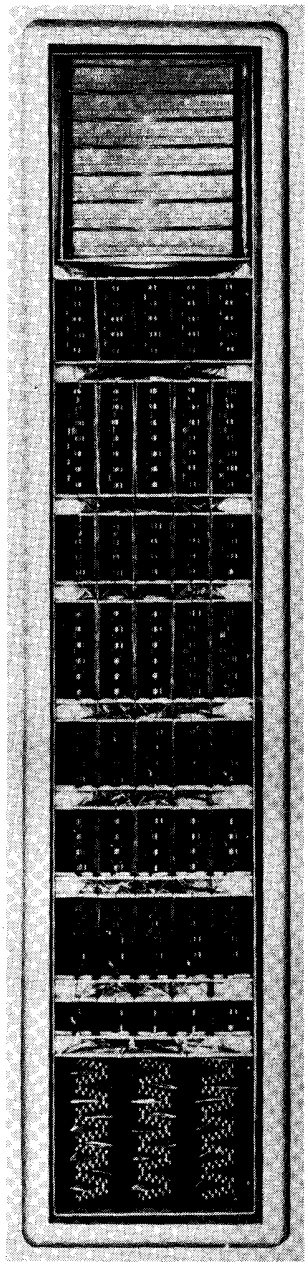
B. Busline System

Every register element consists of a flip-flop which via an input gate can receive information from the buswire according to its position. Likewise information from the flip-flop can be put on the wire via an output gate. Simultaneously sending to, and receiving from, a group of buswires results in a transfer of information from the sending register to the receiving register. The effect of simultaneous sending from more registers depends on the circuits, and will form either the logical sum or the logical product of the contents of the sending registers upon the buslines. In GIER the circuits are chosen so that the logical product is formed.

Connection from the operator's panel to the registers is extremely easy to realize due to the busline system. To indicate and/or to set the contents of a register while the machine is not running, this register must send and receive simultaneously so that its contents are not changed. Indicators on the buswires will now show the bit value, and the register contents can be changed by pressing the set or reset buttons which force the wire to a voltage corresponding to 1 and 0, respectively.

The circuits connecting the flip-flops to the buswires are shown in Fig. 7.

In the neutral position, all send gates are maintained at -3 v and all receive gates at $+5$ v. The buswire is then at $+1.5$ v. When sending, a G_s pulse will open the send gate so that if a 0 is sent, the readout transistor will force the buswire to -1.5 v. In the receiving register element, the read-in transistor, which is symmetrical, will act as an emitter-follower with the collector on the buswire and the emitter coupled to the base at the flip-flop. Thus this is set to 0 and the symmetrical transistor does not saturate. If a one is sent the voltage of the wire is $+1.5$ v, and the symmetrical transistor now acts as a grounded emitter with the emitter on the wire and the collector to the base of the flip-flop, which is thus set to 1. The transistor in this case is driven into saturation and at the end of the gate pulse the stored charge is driven out through the collector and emitter, thus also aiming at setting the flip-flop to 1. The line will remain at a voltage level corresponding to 1, and is not allowed to change until the transistor is out of saturation. The length of the pulses activating the gates is 1 μsec.



End view of the computer showing the microprogram unit.

C. Microprogram Unit

The microprograms are stored in a fixed wired store (reference [5]) built by means of cores of magnetically soft ferrite coupled as current transformers. This type of store is fast and also very cheap. Each store corresponds to a certain microfunction, the secondary winding of the cores being coupled to amplifiers (3-stage amplifiers) which send pulses to the gates, when one of the single turn primary windings is excited by a current. The length of the output pulses is determined directly as the length of the current pulse in the primary winding.

Each wire in the store corresponds to a certain time step in a certain microprogram, and passes through the cores representing all the microfunctions that are to be performed in the present step. (In every step is also specified, as a special microoperation, the step number following in the next cycle.) In total there are about 600 wires.

The selection of the wires is done by means of a simple transistor-diode logic performing voltage coincidence between 1 out of 24 timing flip-flops and 1 out of 68 microprograms. (There is one microprogram per operation, 2 for the address modifications, one for floating-point arithmetic, and one for introducing interrupts from the HP switch.) Conditions are introduced in the microprograms by steering the current through one out of several possible wires.

This technique involves a noise problem, arising when a jump is performed from one micropgrogram to another or when a condition inside the actual microprogram changes, because these situations introduce voltage swings upon groups of wires and this again causes capacitive currents through the wires which may activate the ampliers. In order not to lower the upper frequency limit of the amplifiers for the cores, and thus increase the access time to the store, you have to control the stray capacities between the wires. This is done simply by limiting the number of wires, passing through the same core(s), which may change at the same time, and by controlling the rise and fall times of the voltage swings. As mentioned the cores are coupled as current transformers, so that any wire may pass through as many cores as wanted.

Out of 185 possible microoperations 181 are used (in the expanded GIER system) in the following manner: Simple transfers between registers and buslines, and set and/or reset of one or more flip-flops of a register: 106 microoperations (one amplifier can activate 10 gates as a maximum). Shift 1 or 10 positions left or right: 19 microoperations. Set or reset of flip-flops used as switches in the microprograms including the timing flip-flops for the fixed store: 34 microoperations. And the last group counting, synchronizing, start store cycles, timing of logic circuits (for instance all functions concerning the indicator logic is performed by means of one single microperation): 22 microoperations.

The conditions used in the microprograms are functions of one or more variables. In all, 51 different functions are used, of which 21 are functions of one variable, 18 are functions of a 2 variables, and 12 are functions of 3 or more variables.

As mentioned the fixed wired microprogram store is built as a plug-in unit, being connected to the rest of the machine by means of 20 plugs each with 34 poles. The coupling between the circuits performing the logic functions used as conditions in the microprograms and the drivers for the fixed wired store is pluggable as well. Till now only one set of wiring schemes for the microprogram unit has been developed.

D. The Adder

The adder consists of three parts: A complementer, a sumdigit circuit, and a carry circuit. The two first are built conventionally, while the latter uses the transistor in a special way.[1]

The principle of the carry circuit is seen on Fig. 8. A carry is always formed when both H and MD are 1. A carry should be transmitted from the next lower position to the next higher position, when H and MD are different. If both are zero, a carry should not be transmitted, if they are both one it need not be transmitted, as a carry is generated directly. As seen from Fig. 8, a positive voltage on the terminal $\overline{M}_i$, corresponding to a carry from next lower position, will be transmitted when the function $\overline{MD}\times H+MD\times\overline{H}$ is 1, *i.e.*, negative. The carry will then be transmitted, and it is important that this does not happen when MD = H, because a carry generated in the right transistor would then not only be transmitted to the next higher position but also to the next lower, as the left transistor also conducts in reverse direction. The carry delay in the described circuit is essentially 0. The function $\overline{MD}\times H+MD\times\overline{H}$ is formed in all positions simultaneously while the pulse $\overline{SA}$ is positive, so that no carries are generated at all. In the positions where a subsequent carry should be transmitted, the base of the left transistor is charged by a base current flowing from the −7.5-v clamp through collector and emitter to the base and through the base resistor to −13.5 v. When the transistor bases are charged, the pulse SA becomes negative and the carries are generated. Carries will now be transmitted without delay, as a transistor with the base charged with holes will act as a closed contact as long as the current does not exceed a value corresponding to the base charge.

The time spent for a full 41-bits addition is 2 μsec from the moment where the last addend is placed in its register.

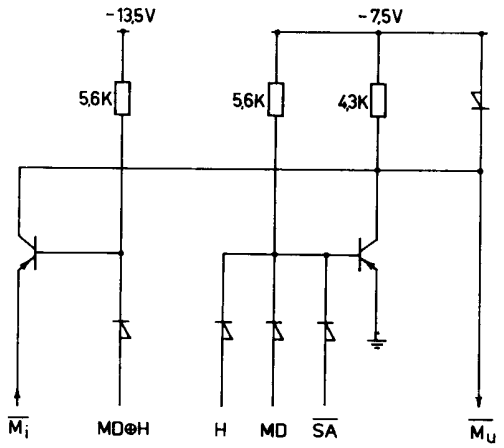


Fig. 8—Carry circuit in the adder.

[1] Similar circuits have been developed by others. See references [2] and [4].

E. Circuit Design Principles

The circuit technique as a whole is based upon individual design, and all circuits are designed to cover the worst case. Complementary circuits are used to some extent. The register elements are built around a conventional flip-flop with transistors driven into saturation. In a few cases an antisaturation circuit has been used. The logical circuits are of the transistor-diode type. The most frequent used transistors are two alloyed junction types with alpha cutoff frequencies of 5 and 9 Mc. Approximately 450 printed circuit cards are used in the standard GIER. The power supply consists of 10 regulated voltages with a power consumption of 500 watts.

VIII. The GIER System

Originally the GIER was constructed as a purely scientific computer system but there soon arose need for a GIER system oriented towards data processing applications and also process control applications. The two first units developed for these purposes are the high-speed printer and the punched card reader. Both of these are built around available mechanical equipment. The ANelex 4-1000 SD printer and the Bull D-3 sorter act as additional one character output/input units. Other units of the GIER system connect via a data channel for transfer of words between the ferrite core store of 1024 words and the block transfer units. The general arrangement of the GIER system is shown in Fig. 6.

A. High-Speed Printer

The mechanical unit is an ANelex 1000-line-per-minute printer to which has been added a two-line corner-turning buffer arranged so that the program sees the printer as it sees the typewriter or the paper tape punch and is operated by the same instructions. The printer is equipped with a two-line buffer so that one line can be filled while another is being printed.

B. Punched Card Reader

The reader is built around the mechanical part of a sorting machine Bull D3. It has two reading stations and 15 sorting magazines. It reads with a nominal speed of 750 cards per minute, but the speed will probably be increased to 1000 cards per minute. The reader is able to read holes as well as pencil marks on the same card.

The sorter is equipped with two sets of brushes BS1 and BS2 (Fig. 9). To read the pencil marks a special circuit is needed, so BS1 is reserved for this purpose, while holes are read with BS2. The information is transferred row by row to the card image buffer, CIB, which is a cornerturning ferrite core buffer. In the intervals between cards the information is read columnwise to a flip-flop register CIF, and simultaneously the corresponding columns of the criterion buffer CRB are read to the flip-flop register CRF. CRB holds the information

of a card and is regenerated while CIB is left reset. The information from CIF and CRF now goes to a set of logic circuits which deliver additional information to go with the information of CIF to the output buffer OUB and control the sorting process in the sorter.

OUB receives the information columnwise and transfers information columnwise to the computer on request.

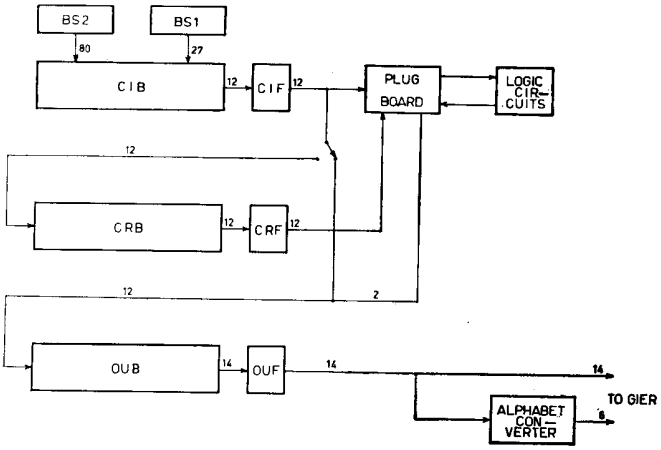


Fig. 9—Communication channels inside the card reader. The digits indicate the number of bits transferred via the path.

C. Buffer Store and Tape Control

The buffer store is a 4096-words ferrite core store, which is equipped with 5 input/output channels, one of which is assigned to the transfer of information to and from the central unit. The four other channels are each assigned to a magnetic tape unit. The buffer will work simultaneously with all five channels according to fixed priorities and can thus be regarded as a subsystem for tape control.

The GIER operations are IL for transfer in the direction towards GIER itself (either from tape to buffer or from buffer to central unit), and US for transfer the other way. The address of the transfer instruction is the number of the data channel (*i.e.*, the wanted unit). Furthermore a parameter word must be placed in the accumulator containing

Beginning address of block in buffer,
Block length in buffer,
Beginning address in the chosen unit,
Block length in the chosen unit.

A block transfer between the chosen unit and buffer takes approximately 12 μsec per word and does not allow the unit to perform other tasks. A block transfer between buffer and tapes is initiated by the central unit, but is controlled by the buffer, so that the central unit can continue the program.

Each tape unit has a two word flip-flop assembly/disassembly register and control circuits specific to its characteristics. So far we have employed two different tape units: the Facit ECM 64 (Caroussel), and the Control Data Corporation 606 tape transport. The ECM 64 is a random access device with 64 spools of tape each holding 8192 words in blocks of 512 words, arranged on a disk so that any spool may be selected within two seconds. The CDC 606 is providing normal tape storage on IBM-compatible tape.

D. Circuit Techniques in the GIER System

While the circuits in the central unit are rather specialized, it has been attempted to use a series of standard circuit blocks in the units of the GIER system. The majority of the circuits are built around a flip-flop, an and gate and two types of inverter cards. This circuit technique is quite conventional and should not be described here.

Of the special circuits used, a single one might be of interest, the circuit for reading information from the brushes of the punched card machine, especially the marks. There seems to be a general tendency to use photoelectric reading, but our experience has shown that satisfactory results can be obtained with brushes with a special "amplifier" circuit.

The pencil marks normally cover 3 brushes, of which the two outside brushes are connected to a positive voltage. The center brush is connected to one plate of a capacitor. At the time a mark can be expected to pass, the capacitor has no charge. During the passage of a mark, a current will pass through the mark between the two brushes and charge the capacitor. At the end of the passage the capacitor is discharged into one side of a normal transistor flip-flop, which is thus set if a mark was present. This kind of amplifier is very sensitive, as the charge necessary to set a flip-flop is only about 3 nc (3×10^{9} coulombs), corresponding to a current of 1 μa for 3 msec. In the actual design the capacitor consists of a coaxial cable, which is necessary anyway to prevent interference between the brushes.

E. Interrupt Unit

As mentioned above the standard GIER is equipped with an interrupt function, initiated by means of the HP switch, especially intended for the operator, so that all manipulations necessary for operating the machine is done by means of this switch and the input typewriter. This HP switch furnishes a pure interrupt feature inclusive masking possibility, but still only from one interrupt channel.

The expanded GIER system has 12 further interrupt channels with an associated masking register. Interrupt calls stopped by the mask are stored in a 12-bits flip-flop register, 1 bit per channel, and will be processed when the mask is opened.

When an interrupt call passes the mask it will cause a jump from the running program to a subroutine which will take care of the interrupt. (If an interrupt occurs

while the machine is not running it may start GIER under some conditions.) During the execution of an interrupt a special flop-flop is set, which will block all further interrupt calls whatever the contents of the masking register are. This flip-flop is reset automatically when the operation pc, place in masking register, is executed, or rather during execution of the instruction immediately after a pc instruction. This feature makes possible interrupt-response programs without the feature of processing interrupts in more than one level. (It should be mentioned that interrupts are not executed until the current instruction has been completed.

When the response routine is finished the machine makes a return jump to the main program, which will continue as if nothing had happened. Concurrent interrupt calls will be processed in a fixed built-in order. The mentioned buffer register for the interrupt channels is not accessible to the programmer except for the possibility of clearing it.

The interrupt system is used for synchronization between GIER and the magnetic tape stations via the 4096 words core store, or real-time processes coupled to GIER via the real-time unit.

F. Real-Time Input-Output Unit (RT Unit)

The RT unit is designed to provide a means of communication between GIER and real-time physical processes, such as analog computers and industrial instrumentation systems. The unit comprises a number of both digital and analog input-output channels. The RT unit may be expanded at will to cover most conceivable applications. In the following, references will be made to Fig. 10 which is a block diagram of an actual RT unit. In addition to serve as a linkage system interconnecting GIER and an analog computer, this particular installation will be used for several research activities in the field of real-time computer control.

All operations pertaining to the data transfers are completely controlled by the computer program on a one word at a time basis. The RT unit communicates with the GIER central processing unit by means of the data channel. Data transfers are released by the read/write instructions IL and US. The transfers are always directed to and from the GIER accumulator register. The RT unit is usually operated in connection with the GIER program interrupt feature for real-time program syncronization.

As shown in Fig. 10, all analog channels and some of the digital ones are made available on a panel equipped with several removable patchboards. Furthermore, all digital channels are terminated on multipin connectors (not shown in figure). As a rule, the RT unit is supplemented with some optional equipment, such as program controllable (via the RT unit) pulse generators to make interrupt signals and dc amplifiers for signal matching purposes.

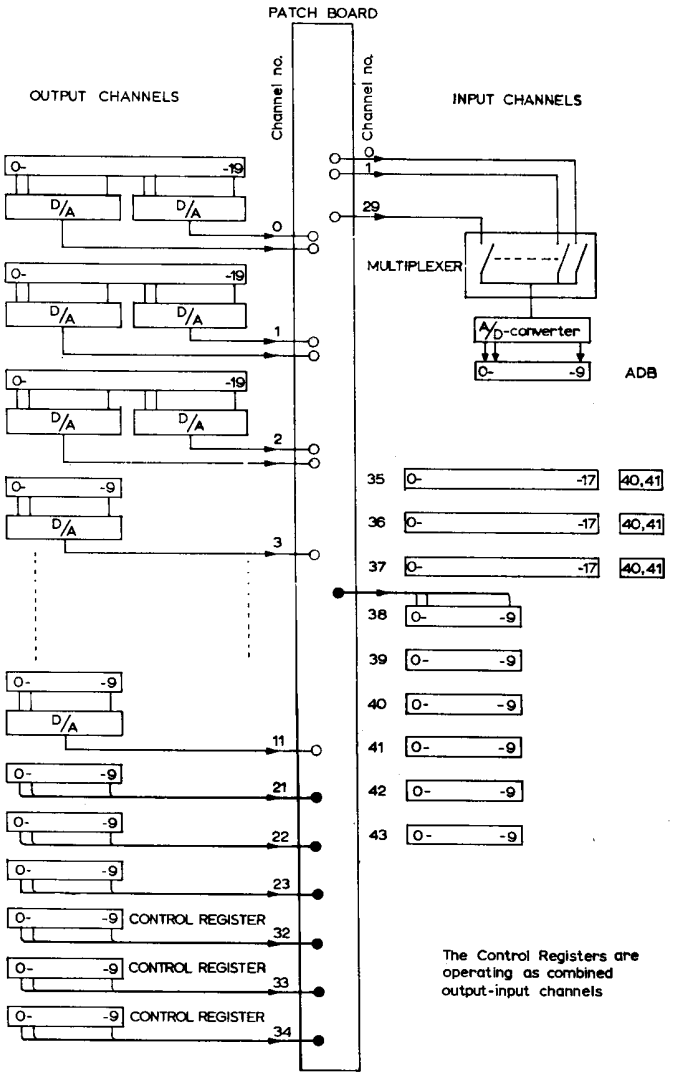


Fig. 10—Block diagram of a typical real-time unit.

Digital channels: Both digital input and output channels are equipped with transistor flip-flop buffer registers for temporary information storage between read/write operations. The buffer registers are available in different lengths up to 42 bits. However, 10 and 20 bits are usual word sizes, communicating with the accumulator bit positions 0–9 and 0–19, respectively. The RT unit channel number is specified by the address part of the IL/US instructions.

The registers referred to as Control Registers in Fig. 10 are recurrently used in subroutines for functional control purposes, such as controlling the operation modes of an analog computer. For the programmer's convenience, these registers have been designed to operate as combined input/output channels. As an additional programming feature data may be loaded into the control registers using part of the accumulator as a mask register.

Analog channels: The analog-to-digital (A/D) and digital-to-analog (D/A) conversion equipment utilizes a 10-bits straight binary code for digital signal representation. Negative numbers are expressed as 2's complements conforming to the usual GIER number interpretation. The choice of word length is partly due to the GIER instruction word format. Considering the corresponding analog signal accuracy, the quantization step equals 0.1 per cent of the signal range which is adequate for most applications.

Sampling of one of the analog input channels is accomplished by letting GIER perform an IL instruction with reference to the particular multiplexer input. The multiplexer gate (Fig. 10) specified by the resultant address of the IL instruction is activated, and the input voltage converted to digital form by the A/D converter and stored in the ADB register. Eventually, the content of ADB arrives in the accumulator. If the input signal exceeds the nominal signal range, the overflow indicator of the GIER accumulator is set.

The analog input signal range is ± 10 v, and the input impedance is 5 kilo-ohms. The actual A/D-conversion time is 50 μsec, and the IL-instruction execution time is 100 μsec.

The analog outputs are generated by D/A converters driven by 10-bits output buffer registers. The buffer register holds the analog output between subsequent write operations—being performed by means of US instructions as explained above—thus, the analog output signal will be a staircase waveform. The no-load D/A-converter output range is ± 5 v and the internal resistance is 2.5 Kilo-ohms. When new information is loaded into the D/A-converter buffer register, the analog output will settle at the new value within 2 μsec. Usually, the analog outputs are applied to the physical process via operational amplifiers for power amplification and signal matching.

As shown in Fig. 10, two D/A converters can be connected to a single 20-bit digital output channel. It is then possible to change the two analog outputs simultaneously, which is an important feature in some applications, for instance when producing x-y oscilloscope displays.

G. The D/A Converter

Fig. 11 shows the principle of the D/A converter that is used in the analog output channels of the RT unit. The change-over switches of Fig. 11 are controlled by the binary positions B0 to B9 of a buffer register, connecting the binary weighted precision resistors either to ground or to the reference power supply buses. Thus, the converter output resistance is constant and equals r/2. Interpreting the contents c of the buffer register by a straight binary code with negative numbers expressed as 2's complement ($-1 \leq c < 1$), we get the converter equivalent circuit shown in Fig. 11.

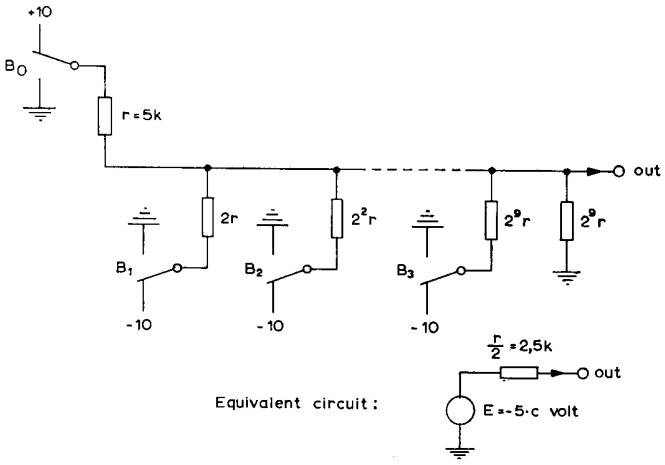


Fig. 11—Principle of D/A converter.

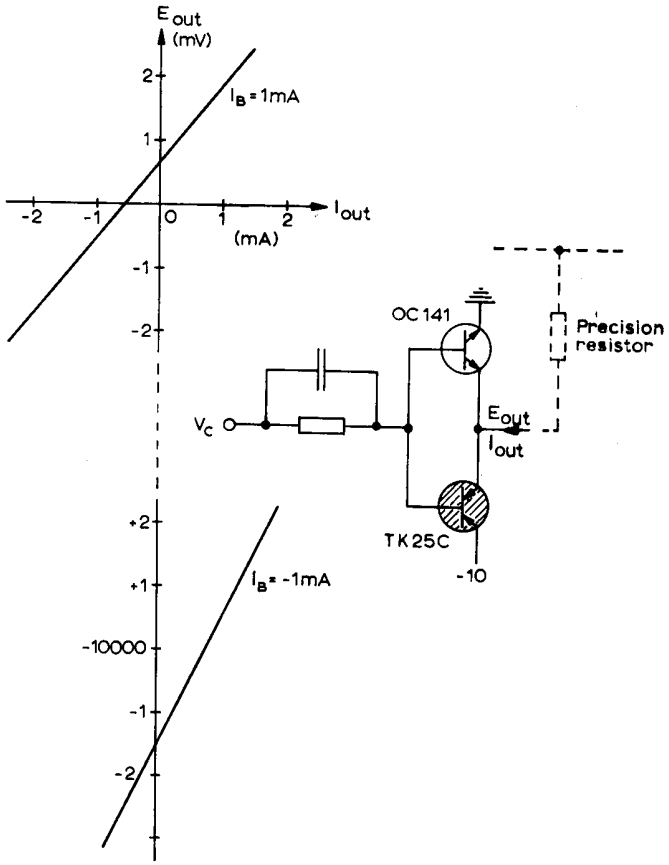


Fig. 12—Complementary emitter-follower switch and characteristics.

The switches of the D/A converter are realized by the complementary emitter-follower circuit shown in Fig. 12. An important feature of the circuit is the fact that the errors caused by leakage currents in the "off" transistors are completely negligible. Both transistors are GE units of symmetrical structure, exhibiting excellent low-level characteristics as saturated switches. The spread of the transistor characteristics is quite small so that no selection of transistors is necessary.

H. The A/D Converter

The multiplexer and the A/D converter are designed as an integrated unit (Fig. 13). When GIER enters an IL instruction with reference to one of the analog input terminals, the particular multiplexer switch is closed, conversion is initiated and the result appears in the ADB register.

The A/D conversion process is controlled by the logical circuitry that adjusts the ADB register for

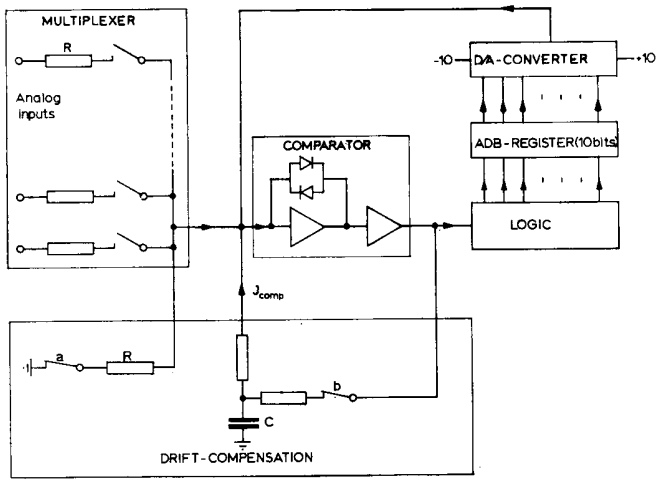


Fig. 13—Schematic block diagram of multiplexer and A/D converter.

current balance at the comparator input. The total conversion time is 50 μsec (5 μsec per bit).

The comparator is realized by a fast dc amplifier operating as an operational amplifier with a nonlinear feedback network (actually consisting of several low-storage, low-capacity GE diodes), as indicated in Fig. 13. The amplifier output is picked up by a fast pulse-forming amplifier that generates the logical output. The zero drift problem is taken care of by a separate drift-compensating circuit. Due to the nonlinear feedback network—reducing the closed-loop gain at high input currents— the amplifier cannot be overdriven and is able to handle the great range of input currents it is subjected to during the signal conversion without loss of recovery speed. The operational amplifier action maintains the input terminal close to ground potential throughout the converting process, *i.e.*, the input impedance is very low. As a consequence, the circuit is quite insensitive to capactive loading at the comparator input. The actual circuit will tolerate a capacitive loading in excess of 1 nF without appreciable deterioration of transient response. This fact allows for the connection of a great number of multiplexer switches directly to the comparator input terminal.

The drift compensation scheme and the multiplexer design will now be described in some detail. The most important drift phenomena are temperature-dependent voltage and current drift of the comparator amplifier and temperature-dependent leakage currents of the multiplexer switches. The drift effects are exactly cancelled by the current I_{comp}, produced by the drift-compensating feedback loop (Fig. 13). The combined gain of the feedback amplifier and the pulse-forming amplifier is very high and negative. The capacitor C ensures closed-loop stability. Switches a and b—which are operated synchronously—are closed and the outer feedback loop activated in between the A/D conversions. Simultaneously the ADB register is zero-set. Switch a is a transistor of the same type and conducting with the same base current as the transistors used in the multiplexer switches, thus simulating a multiplexer input connected to zero. The loop will settle at a value of I_{comp} that balances the comparator so that the logical output operates just on the transition between the two logical output levels. When GIER is executing an IL instruction, switches a and b open and the capacitor C—now operating as a hold circuit—maintains I_{comp} constant during the A/D conversion.

I. The Multiplexer

Considering the design of precision multiplexers equipped with transistor switches, the main difficulty is to supply the base current to keep the switch "closed" without interfering with the signal current to be transmitted. Usually some kind of transformer-coupled drive circuit is employed. In the GIER multiplexer a quite different approach is applied.

The nominal multiplexer input signal range is ±10 v and the input current is ±2 ma (input resistance 5 kilohms). Fig. 14 shows the multiplexer switch in the "open" position: The series switch 2N 2280 is in cutoff and the parallel switch BCZ11 is saturated. The parallel switch (not shown in Fig. 13) maintains a constant multiplexer input impedance. The 1-μf condenser voltage is about 9 v. The switch is put into opera-

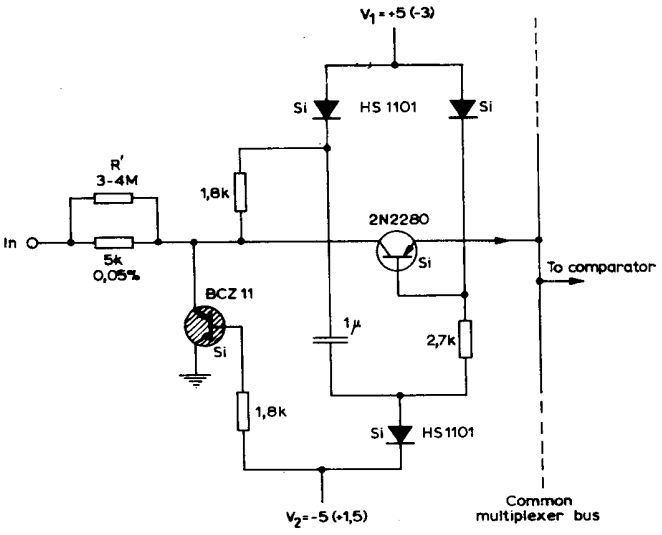


Fig. 14—Multiplexer switch. The switch is shown in the "off" position. Values in brackets indicate voltage levels to drive the switch "on."

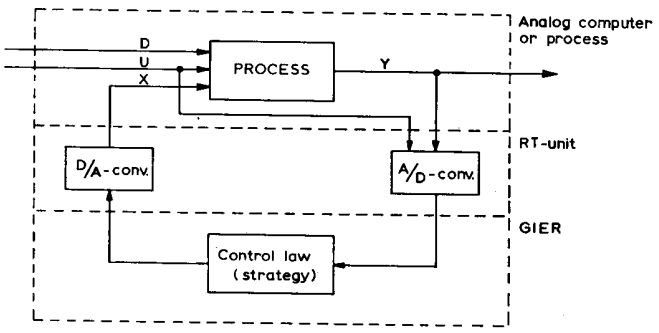
tion by changing the driving voltages V1 from +5 to −3 v and V2 from −5 to +1.5 v. Transistors BCZ11 and diodes HS1101 are cut off and the 1 μf capacitor supplies 2-ma base current to the switch transistor 2N 2280, connecting the multiplexer input resistor to the A/D converter. The saturated 2N 2280 is now floating freely during the A/D conversion, virtually disconnected from the drive circuit (except for negligible leakage currents flowing in the BCZ11 and the HS 1101's, all of which are Si units).

The 2N 2280, which is an Si unit especially designed for low-level switching applications, is conducting in the inverted connection. Resistor R′ (Fig. 14) is adjusted so that the effective multiplexer input resistance is 5 kilo-ohms, the 2N 2280 saturation resistance of 7 ohms included. The voltage offset of the multiplexer switches is balanced by the single 2N 2280-switch (a) of the drift-compensating circuit. The capacitive loading of the comparator input due to each multiplexer switch is about 5 pf.

J. Applications of the Real-Time Unit

Fig. 10 shows a simplified block diagram of an installation running at the Technical University of Norway, Division of Automatic Control.

One of the main research programs at this institution is to study the application of digital computers in process control. Much of the work in this field can profitably be done in the laboratory with the process simulated on an analog computer (which is included in the computer installation). Fig. 15 shows how the different parts work together. Inputs to the process are the controllable input signals and measurable as well as nonmeasurable disturbances. On the basis of the process' outputs and the measurable disturbances, the input variables are calculated according to a certain control law or strategy. With the digital equipment just installed it is the purpose, among other things, to develop and investigate dynamically optimal control strategies.



D—nonmeasurable disturbances
U—measurable disturbances
X—controllable inputs
Y—outputs

Fig. 15—On-line digital computer control.

The realization of these strategies often involves complicated mathematical and logical computations, that makes it necessary to make use of a digital computer.

The time scale in which the analog computer can be run is limited, and sampling periods of 50 msec to 1 sec will be quite common. This puts strict requirements on the input/output equipment and the efficiency with which the computations in GIER go on. The sampling periods can be defined by periodical interrupt signals. A typical interrupt-answer will in this connection be: 1) Read via the RT unit the values of the process' outputs and the measurable disturbances. 2) Calculate on the basis of these the controllable inputs according to a specified strategy. 3) Convert these input values to analog signals and apply them on the process' input terminals.

The different modes of the analog computer (pot. set, IC operate) are controlled by a control register. In order not to occupy special D/A-output channels for setting initial conditions before each run, all the analog output channels from the RT unit can be switched in two groups by relays between two sets of terminals. These relays are controlled by two bits in a control register.

The automatic read-out system of the analog computer is easily controlled from a control register. Analog x-y recorders are connected via the RT unit. The x and y terminals are connected to analog output channels, while control signals, such as operate, stand by, pen up/down are controlled from some bits in a control register. All printing and recording is made as efficient as possible by means of the interrupt system. For example, the printing on typewriter will occupy GIER for about 3 per cent of the time used by the typewriter. In order to make the connection between the analog and digital equipment as flexible as possible, all the input and output channels are coupled to a patchboard as mentioned before. In addition to this the amplifier outputs, the IC terminals of the integrators and different control signals (for instance operation modes, automatic read-out system) from the analog computer are available on this patchboard. It is located in a special interconnecting unit. This unit also contains three floating digital potentiometers which are very useful when working with experimental optimizing control systems and model adjustment. Three control registers of 10 bits are used to realize these potentiometers. By means of relays precision resistors are coupled from one side to the other of "the slider." Two pulse-delay generators and 20 operational amplifiers are also placed in the interconnecting unit. The pulse-delay generators are fully controlled from GIER and among other things they are making periodical interrupt signals for defining the sampling periods. The amplifiers are used for matching the analog signal levels between the RT unit and the analog computer or process. (The analog computer signal range is ±100 v.)

In addition to the laboratory activity, work will be

done in the process industry. GIER together with the RT unit and interconnecting unit will then temporarily be installed in the actual factory. Preliminary investigations will be made for a possibly greater and more specialized process computer control system. Besides being used for control purposes, the computer advantageously can take over a great deal of the supervisory functions in the process, and the data handling and reduction problems. On the basis of the elementary process data the computer is able to calculate and read out data that both really tell something and are up to date. This is almost impossible with analog equipment, but is usually of great importance.

In addition to the project of studying applications of digital computers in process control, there will be great activity in the field of numerically controlled machine tools, using GIER for simulating different types of control-unit schemes.

Acknowledgment

It is a pleasure for the authors to acknowledge the indispensable help received from the many persons who have contributed to the development of GIER, especially T. Krarup from the Geodetic Institute who together with B. Svejgaard designed the list of operations and the microprograms.

References

[1] C. Andersen and C. Gram, "Manual for GIER Programming I and II," Danish Inst. of Computing Machinery, Copenhagen, Denmark; 1962 and 1963 (in Danish, English trans. in press).
[2] Edwards, "Parallel addition in digital computers," *Proc. IEE*, vol. 106B, pp. 464–466; 1959.
[3] P. Naur, "The design of the GIER ALGOL compiler: Part I," BIT, vol. 3, pp. 124–140 and 145–166; 1963.
[4] F. Salter, "High-speed transistorized adder for a digital computer," IRE TRANS. ON ELECTRONIC COMPUTERS, vol. EC-9, pp. 461–464; December, 1960.
[5] Wier, "A high-speed permanent storage device," IRE TRANS. ON ELECTRONIC COMPUTERS, vol. EC-4, pp. 16–20; March, 1955.